Large-Scale Automatic K-Means Clustering for Heterogeneous Many-Core Supercomputer

Yu, Teng; Zhao, Wenlai; Liu, Pan; Janjic, Vladimir; Yan, Xiaohan; Wang, Shicai

Published in:
IEEE Transactions on Parallel and Distributed Systems

DOI:
10.1109/TPDS.2019.2955467

Publication date:
2020

Document Version
Peer reviewed version

Large-Scale Automatic K-Means Clustering for Heterogeneous Many-Core Supercomputer

Teng Yu, Wenlai Zhao, Pan Liu, Vladimir Janjic, Xiaohan Yan, Shicai Wang
Haohuan Fu, Guangwen Yang, John Thomson

Abstract—This paper presents an automatic k-$means$ clustering solution targeting the Sunway TaihuLight supercomputer. We first introduce a multi-level parallel partition approach that not only partitions by dataflow and centroid, but also by dimension, which unlocks the potential of the hierarchical parallelism in the heterogeneous many-core processor and the system architecture of the supercomputer. The parallel design is able to process large-scale clustering problems with up to 196,608 dimensions and over 160,000 targeting centroids, while maintaining high performance and high scalability. Furthermore, we propose an automatic hyper-parameter determination process for k-$means$ clustering, by automatically generating and executing the clustering tasks with a set of candidate hyper-parameter, and then determining the optimal hyper-parameter using a proposed evaluation method. The proposed auto-clustering solution can not only achieve high performance and scalability for problems with massive high-dimensional data, but also support clustering without sufficient prior knowledge for the number of targeted clusters, which can potentially increase the scope of k-$means$ algorithm to new application areas.

Index Terms—Supercomputer, Heterogeneous Many-core Processor, Parallel Computing, Clustering, AutoML

1 Introduction

K-$means$ is a well-known clustering algorithm, used widely in many AI and data mining applications, such as bio-informatics [2], [24], image segmentation [10], [23], information retrieval [37] and remote sensing image analysis [26].

For modern big-data applications, an intelligent clustering solution usually facing two major challenges. Firstly, finding the optimal solution for a general k-$means$ problem is known to be NP-hard [13]. Thus, current high-end k-$means$ applications are limited in terms of the number of dimensions (d), and the number of centroids (k) they can consider, leading to demand for more parallel k-$means$ implementations [3], [26]. Secondly, to determine proper hyper-parameters, such as the targeted number of centroids (k) in k-$means$, are one of the toughest problems especially in newly involved application areas, due to the massive raw data without sufficient prior knowledge for clustering. This also leading to an emerging research topic known as AutoML [**?**].

In this paper, we present an auto-clustering solution based on a supercomputer system. Targeting the above challenges, we make the following two main contributions in our work.

We propose a novel parallel design of k-$means$ with multi-level partition targeting Sunway TaihuLight, one of the world's fastest supercomputers. This design allows k-$means$ to scale well across a large number of computation nodes, significantly outperforming previously proposed techniques. Evaluation results show that the proposed design is able to process large-scale clustering problems with up to 196,608 dimensions and 160,000 centroids, while maintaining high performance and scalability, which is a large improvement on previous implementations, as described in Table 1.

Furthermore, we propose and implement an auto-clustering process based on the parallel algorithm design, including four new features: a) a task generator to automatically generate clustering tasks according to a number of candidate hyper-parameters; b) a self-aware method to do the automatic dataflow partition for the generated tasks; c) a fairness resource allocator with a task scheduler to launch the clustering tasks to the supercomputer system; d) an evaluation method to determine the best hyper-parameter candidate based on the clustering results.

With a highly scalable algorithm design and an automatic hyper-parameter determination process, our method can greatly increases the potential scope for k-$means$ applications to solve previously intractable problems.

The rest of this paper is organized as follows: Section 2 describes the background and related work which includes a short description of Sunway supercomputer and the k-$means$ problem definition, the most popular Lloyd algorithm and general parallel implementation, and the state-of-the-art supercomputer-oriented designs in the literature. Section 3 discusses the three levels scalable design and implementation of k-$means$ on Sunway. Section 4 discusses the auto-clustering process design and implementation. Evaluation results and analysis are given in Section 5.

- *T.Yu, V.Janjic and J.Thomson with University of St Andrews, UK. E-mail: {ty33,vj32,j.thomson}@st-andrews.ac.uk*
- *W.Zhao, P.Liu, H.Fu and G.Yang with Tsinghua University and National Supercomputer Centre in Wuxi, China. W.Zhao is the corresponding author. E-mail: {zhaowenlai,liupan15,haohuan,ygw}@tsinghua.edu.cn*
- *S.Wang is with Wellcome Trust Sanger Institute, UK. E-mail: sw23@sanger.ac.uk*
- *X.Yan is with University of California, Berkeley, US. E-mail: xiaohan_yan@berkeley.edu*

TABLE 1
Parallel k-$means$ Implementations

Approaches	Hardware	Programming model	Samples n	Clusters k	Dimensions d
General Parallel k-$means$ Implementations					
Böhm, et al [5]	Multi-core	MIMD/SIMD	10^7	40	20
Hadian and Shahrivari [19]	Multi-core	multi-thread	10^9	100	68
Zechner and Granitzer [40]	GPU	CUDA	10^6	128	200
Li, et al [27]	GPU	CUDA	10^7	512	160
Haut, et al [21]	Cloud	OpenStack	10^8	8	58
Cui, et al [11]	Cluster	Hadoop	10^5	100	9
Supercomputer-Oriented k-$means$ Implementations					
Kumar, et al [26]	$Jaguar$, Oak Ridge	MPI	10^{10}	1000	30
Cai, et al [7]	$Gordon$, SDSC	mclappy (parallel R)	10^6	8	8
Bender, et al [3]	$Trinity$, NNSA	OpenMP	370	18	140,256
Our approach	$Sunway$, **Wuxi**	**DMA/MPI**	10^6	**160,000**	**196,608**

2 Background

2.1 Sunway TaihuLight and SW26010 Many-Core Processor

Sunway TaihuLight is a world-leading supercomputer, which currently ranks as the third in the TOP500 list [29] and achieves a peak performance of 93 petaflops [17].

Sunway TaihuLight uses the SW26010 many-core processor. The basic architecture of SW26010 is shown in Figure 1. Each processor contains four *core groups (CGs)*. There are 65 cores in each CG, 64 *computing processing element (CPEs)* and a *managing processing element (MPE)*, which are organized as 8 by 8 mesh. The MPE and CPE are both complete 64-bit RISC cores, but they are assigned different tasks while computing. The MPE is designed for management, task schedule, and data communications. The CPE is assigned to maximize the aggregated computing throughput while minimize the complexity of the micro-architecture.

The SW26010 design differs significantly from the other multi-core and many-core processors: (i) for the memory hierarchy, while the MPE applies a traditional cache hierarchy (32-KB L1 instruction cache, 32-KB L1 data cache, and a 256-KB L2 cache for both instruction and data), each CPE only supplies a 16-KB L1 instruction cache, and depends on a 64 KB *Local directive Memory (LDM)* (also known as *Scratch Pad Memory (SPM)*) as a user-controlled fast buffer. The user-controlled 'cache' leads to some increasing programming difficulties for using fast buffer efficiently, at the same time, providing the opportunity to implement a defined buffering scheme which is beneficial to improve the whole performance in certain cases. (ii) As for the internal information of each CPE mesh, we have a control network, a data transfer network (connecting the CPEs to the memory interface), 8 column communication buses, and 8 row communication buses. The 8 column and row communication buses provide possibility for fast register communication channels to across the 8 by 8 CPE mesh, so users can attain a significant data sharing capability at the CPE level.

2.2 Problem Definition

The purpose of the k-means clustering algorithm is to find a group of clusters to minimize the mean distances between samples and their nearest centroids. Formalized, given n samples, $\mathcal{X}^d = \{x_i^d \mid x_i^d \in R^d, i \in \{1, ..., n\}\}$, where each sample is a d-dimensional vector $x_i^d = (x_{i1}, \ldots, x_{id})$ and we

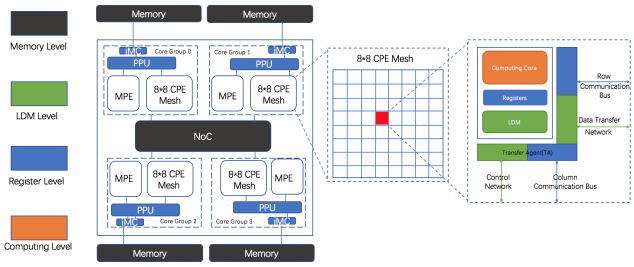


Fig. 1. The general architecture of the SW26010 many-core processor

use u to index the dimensions: $u \in \{1 \ldots d\}$. We aim to find k d-dimensional centroids $\mathcal{C}^d = \{c_j^d \mid c_j^d \in R^d, j \in \{1 \ldots k\}\}$ to minimize the object $\mathcal{O}(\mathcal{C})$:

$$\mathcal{O}(\mathcal{C}) = \frac{1}{n} \sum_{i=1}^{n} \text{dis}(x_i^d, c_{a(i)}^d)$$

Where $a(i) = arg\ min_{j \in \{1 \ldots k\}} \text{dis}(x_i^d, c_j^d)$ is the index of the nearest centroid for sample x_i^d, $dis(x_i^d, c_j^d)$ is the $Euclidean$ distance between sample x_i^d and centroid c_j^d:

$$\text{dis}(x_i^d, c_j^d) = \sqrt{\sum_{u=1}^{d} (x_{iu} - c_{ju})^2}$$

In the literature, several methods have been proposed to find efficient solutions [6], [12], [15], [31], [32], [35]. While the most popular baseline is still the $Lloyd$ algorithm [30], which is composed by repeating the basic two steps below:

$$1.: \ a(i) = arg\ min_{j \in \{1 \ldots k\}}\ dis(x_i^d, c_j^d)\ (Assign)$$

$$2.: \ c_j^d = \frac{\sum_{arg\ a(i)=j} x_i^d}{|arg\ a(i) = j|} \quad (Update)$$

We also need to chose an initial set of centroids. Note that those notations here are mainly from previous works by Hamerly [20], Newling and Fleuret [31]. We will apply customized notations only when needed. The first step above is to assign each sample into the nearest centroid according to the $Euclidean$ distance. The second step is to update the centroids by moving them to the mean of their assigned samples in the d-dimensional vector space. Those two steps are repeated until each c_j^d is fixed.

2.3 Related Works

2.3.1 General Parallel k-means

k-means algorithm has been widely implemented in parallel architectures with shared and distributed memory using either SIMD or MIMD model targeting on multi-core processors [5], [14], [19], GPU-based heterogeneous systems [27], [38], [40], clusters of computer/cloud [11], [21].

In the parallel case, we use l to index the processors (computing units) $\mathcal{P}$ ($\mathcal{P} = \{P_l\}, l \in \{1 \dots m\}$), and use m to denote the total number of processors applied. The dataset $\mathcal{X}^d$ is partitioned uniformly into m processors. Compared with $Lloyd$ algorithm, each processor is assigns a subset ($\frac{n}{m}$) of samples from the original set $\mathcal{X}^d$ before the $Assign$ step.

To facilitate communication between computing units, the Message Passing Interface (MPI) library is mostly applied in common multi-core processor environments. Performance nearly linearly increases with the limited number of processors as the communication cost between processes can be ignored in the non-scalable cases, as demonstrated in [14]. Similarly, the $Update$ steps are finished by m processors in parallel through MPI as well. Processors should communicate with each other before the final c_j^d can be updated.

2.3.2 Large-scale Parallel k-means on Supercomputers

In addition to general parallel k-means implementations, other customized k-means implementation targeting on supercomputers are more related to our work here.

Kumar, et al [26] implemented the dataflow-partition based parallel k-means on the $Jaguar$, a Cray XT5 supercomputer at Oak Ridge National Laboratory evaluated by real-world geographical datasets. Their implementation applys MPI protocols to achieve broadcasting and reducing and originally scaled the value of k to more than 1,000s level.

Cai, et al [7] designed a similar parallel approach on $Gordon$, a Intel XEON E5 supercomputer at San Diego Supercomputer Center for grouping game players. They applied a parallel R function, $mclapply$, to achieve shared-memory parallelism and test different degree of parallelism by partitioning the original data-flow into different numbers of sets. They did not focus on testing the scalability of their approach but evaluated on the quality of the cluster.

Bender, et al [3] investigated a novel parallel implementation proposed for $Trinity$, the latest National Nuclear Security Administration supercomputer with Intel Knight's Landing processors and their $scratchpad$ two-level memory model. Their approach is the most state-of-the-art comparable work against our proposed methods which can not only partition dataflow, but also partition the number of target clusters k by their $hierarchical$ two-level memory support - cache associated with each core and $scratchpad$ for share. Adapted originally from [18], their partitioning algorithm partitioned the input dataset into $\frac{nd}{M}$ sets, where M is the size of the $scratchpad$, and then reduced $k\frac{nd}{M}$ centroids recursively if needed. Based on this partition, their approach scaled d into 100,000s level.

A fundamental bottleneck in their approach is that based on only two-level memory, it is still impossible to partition and then scale both k and d independently. This leads to the interaction constraint between k and d as discussed in their paper:

$$Z < kd < M$$

where Z is the size of cache. This partition-based method is not efficient if all k centroids could fit into one cache. In practice, this limits the value of k to be less than 18 and d to be greater than 152,917 in their experiments. We claim that our proposed approach with underlining data partitioning methods based on hierarchical many-core processors achieves the needed multi-level fully nkd partition with architectural support to thoroughly solve this bottleneck.

We formalize the background work of both general parallel k-means and supercomputer-oriented implementations as shown in Table 1.

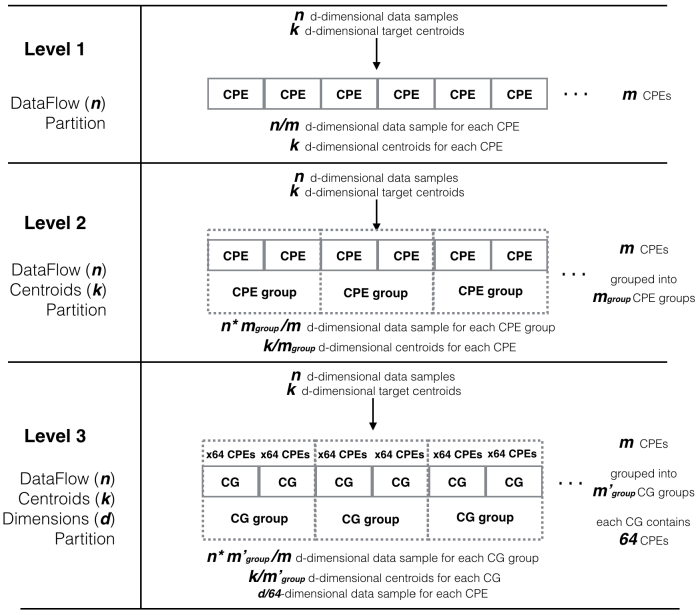


Fig. 2. Three-level k-means design for data partition and parallelism on Sunway architecture

3 Multi-level large-scale k-means design

The scalability and performance of parallel k-means algorithm on large-scale heterogeneous systems and supercomputers are mainly bounded by the memory and bandwidth. To achieve efficient large-scale k-means on the Sunway supercomputer, we explore the hierarchical parallelism on our heterogeneous many-core architecture. We demonstrate the proposed scalable methods on three parallelism levels by how we partition the data.

- Level 1 - $DataFlow$ Partition: Store a whole sample and k centroids on single-CPE
- Level 2 - $DataFlow$ and $Centroids$ Partition: Store a whole sample on single-CPE whilst k centroids on multi-CPE
- Level 3 - $DataFlow$, $Centroids$ and $Dimensions$ Partition: Store a whole sample on multi-CPE whilst k centroids on Multi-CG and d dimensions on Multi-CPE

An abstract graph of how we partition the data into multiple levels is presented in Figure 2.

3.1 Level 1 - DataFlow Partition

Algorithm 1 Basic Parallel k-$means$

1: **INPUT:** Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
2: $P_l \xleftarrow{load} \mathcal{C},\ l \in \{1 \dots m\}$
3: **repeat**
4: // Parallel execution on all CPEs:
5: **for** $l = 1$ *to* m **do**
6: Init a local centroids set $\mathcal{C}^l = \{c_j^l | c_j^l = \mathbf{0}, j \in [1, k]\}$
7: Init a local counter $count^l = \{count_j^l | count_j^l = 0, j \in [1, k]\}$
8: **for** $i = (1 + (l-1) * \frac{n}{m})$ *to* $(l * \frac{n}{m})$ **do**
9: $P_l \xleftarrow{load} x_i$
10: $a(i) = arg\ min_{j \in \{1 \dots k\}} dis(x_i, c_j)$
11: $c_{a(i)}^l = c_{a(i)}^l + x_i$
12: $count_{a(i)}^l = count_{a(i)}^l + 1$
13: **end for**
14: **for** j = 1 *to* k **do**
15: AllReduce c_j^l and $count_j^l$
16: $c_j^l = \frac{c_j^l}{count_j^l}$
17: **end for**
18: **end for**
19: **until** $\mathcal{C}^l == \mathcal{C}$
20: **OUTPUT:** $\mathcal{C}$

In the simple case, we run the first step, $Assign$, on each CPE in parallel while using multi-CPE collaboration to implement the second step, $Update$. The pseudo code of this case is shown in Algorithm 1.

The $Assign$ step is implemented similarly to the traditional parallel k-$means$ algorithm – (1.1) and (1.2) as above. Given n samples, we partition into multiple CPEs. Each CPE (P_l) firstly reads one sample x_i and finds the minimum distances dis from itself to all centroids c_j to obtain $a(i)$. Then two variables are accumulated for each cluster centroid c_j according to $a(i)$, shown in line 11 and 12. The first variable stores the vector sum of all the samples assigned to c_j, notated as $c_{a(i)}^l$. The second variable counts the total number of samples assigned to c_j, notated as $count_{a(i)}^l$.

In the $Update$ step, we first accumulate the c_j^l and $count_j^l$ of all CPEs by performing two AllReduce operations, so that all CPEs can obtain the assignment results of the whole input dataset. We use $register\ communication$ **[?]** to implement intra-CG AllReduce operation and use MPI_AllReduce for inter-CG AllReduce. After the accumulation, the $Update$ step is performed to calculate new centroids, as shown in line 15.

Analysis

Considering a one-CG task, we analyse the constraints on scalability in terms of memory limitation of each CPE. Based on the steps above, one CPE has to accommodate at least one sample x_i, all cluster centroids $\mathcal{C}$, k centroids' accumulated vector sum $\mathcal{C}^l$ and k centroids' counters $count^l$. Considering that each CPE has a limited size of LDM, we obtain the constraint ($\mathbf{C}_1$) below:

$$\mathbf{C}_1: \quad d(1 + k + k) + k \leq LDM$$

Since both the number of centroids k and the dimension d for each sample x_i should at least be 1, we obtain two more boundary constraints ($\mathbf{C}_2$) and ($\mathbf{C}_3$) below, separately:

$$\mathbf{C}_2: \quad 3d + 1 \leq LDM$$

$$\mathbf{C}_3: \quad 3k + 1 \leq LDM$$

Now we analyse the performance under bandwidth bounds. Note that the $Assign$ step of computing $a(i)$ for each sample x_i is completed fully in parallel on the m CPEs. Given the bandwidth of multi-CPE architecture to be B, the DMA time of reading data from main memory can be simply formalized as:

$$\mathbf{T}_{read}: \quad (\frac{n * d}{m} + k * d)/B$$

Theoretically, a linear speedup for computing time to at most n times against the serial implementation can be obtained for the $Assign$ step if we can apply $m = n$ CPEs in total.

The two AllReduce operations are the bottleneck process in the $Update$ step. The $register\ communication$ technique for internal multi-CPE communication guarantees a high-performance with a normally 3x to 4x speedup than other on-chip and Internet communication techniques (such as DMA and MPI) for this bottleneck process (referring to the experimental configuration section for detailed quantitative values). Given the bandwidth of $register\ communication$ to be R, the time for the AllReduce process can be formalized as:

$$\mathbf{T}_{comm}: \quad \frac{n}{m}((1 + k) * d)/R$$

3.2 Level 2 - DataFlow and Centroids Partition

To scale the number of k for cluster centroids $\mathcal{C}$, we use multiple (up to 64) CPEs in one CG to partition the set of centroids. The number of CPEs grouped to partition the centroids is denoted by m_{group}. For illustration, we use l' to index the CPE groups $\{\mathrm{P}\}$. Then we have:

$$\{P\}_{l'} := \{P_l\},\ l \in (1 + (l' - 1) * m_{group}, l' * m_{group})$$

The pseudo code of this case is shown in Algorithm 2. To partition k centroids on m_{group} CPEs, we need to do a new sub-step against the previous case as shown in line 2. Then different from the $Assign$ step in above case, we partition each data sample x_i in each CPE group as shown in line 8. After that, similar to (1.2), all P_l in each $\{P\}_{l'}$ can still compute a partial value of $a(i)$ (named as $a(i)'$) fully in parallel without communication. Note that the domain of j in line 11 is only a subset of $(1, \dots, k)$ as presented above in line 2, so we need to do one more step by data communication between CPEs in each CPE group to obtain the final a(i) as shown in line 10.

Then the $Update$ step is similar to previous case. We just view one CPE group as one basic computing unit, which conducts what a CPE did in the previous case. Each CPE only computes values of subset of centroids $\mathcal{C}$ and does not need further communications in this step as it only needs to store this subset.

Algorithm 2 Parallel k-$means$ for k-scale

1: **INPUT:** Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
2: $P_l \xleftarrow{load} c_j \; j \in (1 + mod(\frac{l-1}{m_{group}}) * \frac{k}{m_{group}}, \; (mod(\frac{l-1}{m_{group}}) + 1) * \frac{k}{m_{group}})$
3: **repeat**
4: // Parallel execution on each CPE group $\{P\}_{l'}$:
5: **for** $l' = 1$ *to* $\frac{m}{m_{group}}$ **do**
6: Init a local centroids set $\mathcal{C}^{l'}$ and counter $count^{l'}$
7: **for** i = $(1 + (l'-1)\frac{n*m_{group}}{m})$ *to* $(l'\frac{n*m_{group}}{m})$ **do**
8: $\{P\}_{l'} \xleftarrow{load} x_i$
9: $a(i)' = arg\ min_j\ dis(x_i, c_j)$
10: $a(i) = min.\ a(i)'$
11: $c^{l'}_{a(i)} = c^{l'}_{a(i)} + x_i$
12: $count^{l'}_{a(i)} = count^{l'}_{a(i)} + 1$
13: **end for**
14: **for** j = $(1 + mod(\frac{l-1}{m_{group}}) * \frac{k}{m_{group}})$ *to* $((mod(\frac{l-1}{m_{group}}) + 1) * \frac{k}{m_{group}})$ **do**
15: AllReduce $c^{l'}_j$ and $count^{l'}_j$
16: $c^{l'}_j = \frac{c^{l'}_j}{count^{l'}_j}$
17: **end for**
18: **end for**
19: **until** $\cup\, \mathcal{C}^{l'} == \mathcal{C}$
20: **OUTPUT:** $\mathcal{C}$

Analysis

To analyse the scalability of k in this case, the amount of original k centroids distributed in m_{group} CPEs leads to a easier constraint of k against the ($\mathbf{C}_3$) above:

$$\mathbf{C'}_3 :\ 3k + 1 \leq m_{group} * LDM\ (m_{group} \leq 64)$$

Based on this, we can also easily scale the ($\mathbf{C}_1$) as follow:

$$\mathbf{C'}_1 :\ d(1 + k + k) + k \leq m_{group} * LDM\ (m_{group} \leq 64)$$

Note that we still need to accommodate at least one d-dimensional sample in one CPE, so the ($\mathbf{C}_2$) should be kept as before: $\mathbf{C'}_2 := \mathbf{C}_2$

As for performance, since m_{group} CPEs in one group should read the same sample simultaneously, the processors need more time to read the input data samples than the first case, but only partial cluster centroids need to be read by each CPE:

$$\mathbf{T'}_{read} :\quad (\frac{n * d * m_{group}}{m} + \frac{k}{m_{group}} * d)/B$$

As for the data communication needed, there is one more bottleneck process (line 12) than before. Comparing against the above cases, multiple CPE groups can be allocated in different processors. Those communication need to be done through MPI which is much slower than internal processor multi-CPEs $register\ communication$. Given the bandwidth of network communication through MPI to be M, we obtain:

$$\mathbf{T'}_{comm} : \frac{k}{m_{group}}/R + \frac{n * m_{group}}{m}((1 + k) * d))/M$$

Algorithm 3 Parallel k-$means$ for k-scale and d-scale

1: **INPUT:** Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
2: $CG_{l''} \xleftarrow{load} c^d_j,\ l'' \in \{1 \dots \frac{m}{64}\}, j \in (1 + mod(\frac{l''-1}{m'_{group}}) * \frac{k}{m'_{group}}, \; (mod(\frac{l''-1}{m'_{group}}) + 1) * \frac{k}{m'_{group}})$
3: **repeat**
4: // Parallel execution on each CG group $\{CG\}_{l''}$:
5: **for** $l'' = 1$ *to* $\frac{m}{64}$ **do**
6: Init a local centroids set $\mathcal{C}^{l''}$ and counter $count^{l''}$
7: **for** i = $(1 + (l''-1)\frac{n*m'_{group}}{m})$ *to* $(l''\frac{n*m'_{group}}{m})$ **do**
8: **for** u = $(1 + mod(\frac{l-1}{64}) * \frac{d}{64}$ *to* $(mod(\frac{l-1}{64}) + 1) * \frac{d}{64})$ **do**
9: $CG_{l''} \leftarrow x_i\ (P_l \leftarrow x^u_i)$
10: **end for**
11: $a(i)' = arg\ min_j\ dis(x_i, c_j)$
12: $a(i) = min.\ a(i)'$
13: $c^{l''}_{a(i)} = c^{l''}_{a(i)} + x_i$
14: $count^{l''}_{a(i)} = count^{l''}_{a(i)} + 1$
15: **end for**
16: **for** j = $(1 + mod(\frac{l''-1}{m'_{group}}) * \frac{k}{m'_{group}})$ *to* $((mod(\frac{l''-1}{m'_{group}}) + 1) * \frac{k}{m'_{group}})$ **do**
17: AllReduce $c^{l''}_j$ and $count^{l''}_j$
18: $c^{l''}_j = \frac{c^{l''}_j}{count^{l''}_j}$
19: **end for**
20: **end for**
21: **until** $\cup\, \mathcal{C}^{l''} == \mathcal{C}$
22: **OUTPUT:** $\mathcal{C}$

3.3 Level 3 - DataFlow and Centroids and Dimensions Partition

To scale the number of dimension d for each sample x_i and further scale k, we store and partition one d-dimensional sample by one CG with 64 CPEs and then implement the algorithm on multiple CGs. The pseudo code of this case is shown in Algorithm 3.

Recall we use u to index the data dimension: $u \in (1 \dots d)$; Now we use l'' to index the CGs and m'_{group} to denote the number of CGs grouped together to partition k centroids. Consider that we apply m CPEs in total and each CG contains 64 CPEs, then we have $l'' \in (1, \dots, \frac{m}{64})$, $m'_{group} \leq \frac{m}{64}$ and:

$$CG_{l''} := \{P_l\},\ l \in (1 + 64(l'' - 1),\ 64l'')$$

To partition k centroids on multiple CGs, we obtain an updated step against the previous case as shown in line 2. To partition each d-dimensional sample x^d_i on 64 CPEs in one CG, we obtain the following step as shown in line 9.

Similar to the above case, all $CG_{l''}$ in each CG group compute the partial value $a(i)'$ fully in parallel and then communicate to obtain the final $a(i)$. Multi-CG communication in multiple many-core processors (nodes) is implemented through MPI interface. Then the $Update$ step is also similar to the previous case. Now we view one CG as one basic computing unit which conducts what one CPE did before and we view what a CG group does as what a CPE group did before.

Analysis

In this case, each CG with 64 CPEs accommodates one d-dimensional sample x_i. Then we can scale the previous ($\mathbf{C}_2$) as follow:

$$\mathbf{C''}_2 : \ 3d + 1 \leq 64 * LDM$$

Consider we use totally m'_{group} CGs to accommodate k centroids in this case, then ($\mathbf{C}_3$) will scale as follow:

$$\mathbf{C''}_3 : \ 3k + 1 \leq m'_{group} * 64 * LDM$$

Note that the domain of m'_{group} seems limited by the total number of CPEs applied, m. But in fact, this number can be large-scale as we target on the supercomputer with tens of millions of cores. Finally, ($\mathbf{C}_1$) will scale as follow:

$$\mathbf{C''}_1 : \ d(1 + k + k) + k \leq 64 * m'_{group} * LDM$$

which is equal to:

$$\mathbf{C''}_1 : \ d(1 + k + k) + k \leq m * LDM$$

$\mathbf{C''}_1$ is the breakthrough contribution over other state-of-the-art work [3]: the total amount of $d * k$ is not limited by a single or shared memory size any more. It is fully scalable by the total number of processors applied (m). In a modern supercomputer, this value can be large-scaled up-to tens of millions when needed.

Considering performance, note that m'_{group} CGs (64 CPEs in each) in one group should read the same sample simultaneously. In another aspect, each CPE only needs to read a partial of the given d-dimension of original data sample together with a partial of k centroids similarly as before, then we obtain a similar reading time:

$$\mathbf{T''}_{read} : \quad (\frac{n * d * m'_{group}}{m} + \frac{k}{m'_{group}} * \frac{d}{64})/B \qquad (1)$$

Comparing against the above cases, multiple CGs in CG groups allocated in different many-core processors need communication to update centroids through MPI. Given the bandwidth of network communication through MPI to be M, the cost between multiple CG groups can be formalized as:

$$\mathbf{T''}_{read} : \quad (\frac{n * d * m'_{group}}{m} + \frac{k}{m'_{group}} * \frac{d}{64})/B \qquad (2)$$

The network architecture of Sunway TaihuLight is a two-level fat tree. 256 computing nodes are connected via a customized inter-connection board, forming a *super-node*. All super-nodes are connected with a central routing server. The intra super-node communication is more efficient than the inter super-node communication. Therefore, in order to improve the overall communication efficiency of our design, we should make a CG group located within a super-node if possible.

4 Auto-clustering Process

Based on the parallel k-means design, we further propose an auto-clustering process to determining the optimal hyper-parameter (k) for applications that is lack of prior clustering knowledge. The key idea is that we can run the clustering with a set of candidate hyper-parameters, and then provide a method to evaluate the best candidate hyper-parameter(s) based on the clustering results.

We first described the method to determine the optimal hyper-parameter (k) for the k-means algorithm on a given input set. Then we introduce our design to solve two practical problems - how to automatically select the data partitioning method to process the workload when the value of k changes, and how to allocate resources of a supercomputer for different instances of the k-means algorithm.

4.1 Determinig the optimal k

The number of clustering (k) need to be predetermined for typical k-$means$ algorithms. As claimed in the survey [41], how to define this value is an critical question in the community, and inappropriate decision would yield poor quality of clustering results.

Shi, et al. [36] proposed a basic method by gradually increasing the possible number of clusters and used the result when the distortion of solutions between current k and k-1 is less than a static predefined threshold. Chen, et al. [9] recently presented a method without any predefined threshold. It generates a formula by computing the difference between sum of distance inside and outside clusters.

While this formulation didn't work in large-scale cases as it keeps monotonous increasing when the k is greater than 2.

To solve this problem with a supercomputing-based approach, we introduce the notion of cluster radius $r(k)$ to k-$means$ clustering. To be specific, $r(k)$ is defined to be the smallest non-negative real number such that the sample set $\mathcal{X}^d$ can be covered by k closed balls centered at sample points with radius $r(k)$. In other words,

$$r(k) = \inf\{t : \exists y_1, \ldots, y_k \text{ in } R^d, \mathcal{X}^d \subseteq \bigcup_{1 \leq s \leq k} B(y_s, t)\},$$

where $B(y_s, t)$ stands for the Euclidean closed ball centered at y_s with radius t. For instance, when $k = n$ the number of samples, we have $r(n) = 0$. It is easy to see that $r(k)$ is non-increasing with respect to k. Radius has been widely used in clustering problems, such as approximating clustering [1] and incremental clustering [8], but not on k-$means$, because it is impossible to compute and measure all possible radius values on large-scale datasets. For n samples clustering into k centroids, there will be $O(n^k)$ possible solutions.

With the support of modern supercomputer with efficient parallel processing techniques, we apply an empirical way by using a minim radius from a random selection of solutions with k centroids, named $r'(k)$ to represent the $r(k)$. With the increasing of k, the accurate of $r'(k)$ will decrease. The $r'(k)$ will even increase at some point when it is too difficult to give a show a reasonable representation of $r(k)$ by $r'(k)$ from a limited selection of solutions. This also indicates that to keep increasing the targeted centroids (k) beyond this points becomes meaningless as it cannot easily reduce the distance and distinguish the difference from different clusters. So the idea of determining the best k is by measuring the change of $r'(k)$ with respect to $r(k)$. If $r'(k)$ does not keep the same route of $r(k)$, we would regard this k as a satisfying choice. Rigorously speaking, let

$$\Delta r'(k) = r'(k) - r'(k + 1),$$

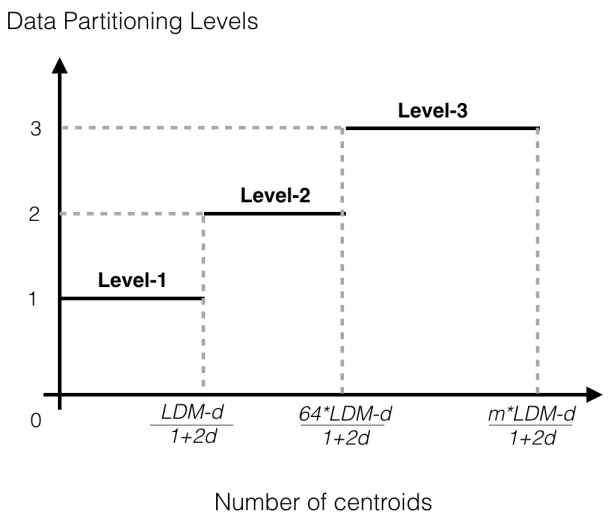


Fig. 3. Self-aware Roofline Model for Auto Data Partitioning

then our optimal k is taken as the first time this function $\Delta r'$ increasing.

4.2 Self-aware Auto Dataflow Partition

A self-aware method to auto partition dataflow into 3 levels based on the targeting k values. This method is mainly guided by the scalability of each level of data partitioning. Based on the limitations presented in formulations $(\mathbf{C}_1, \mathbf{C'}_1, \mathbf{C''}_1)$ above, we can easy compute the range of possible k values for each level: $k \leq \frac{LDM-d}{1+2d}$ for level-1, $k \leq \frac{64LDM-d}{1+2d}$ for level-2 and $k \leq \frac{m*LDM-d}{1+2d}$ for level-3. By concatenating the ranges, we obtain the self-aware 3-stage roofline model to guide the data partitioning as shown in figure 3.

4.3 Fairness Resource Allocation and Task Scheduler

Algorithm 4 Resource Allocation Algorithm

1: **INPUT:** Cost function $T''(k, m, m')$, number of CPEs p, number of CPEs per CG q, number of points n
2: **for** i=1 to $n/2$ **do**
3: $m_i = q$; $m'_i = 0$
4: **end for**
5: remProc = $p - (n/2) * q$
6: **while** remProc > 0 **do**
7: $i = \text{argmin}_{j=1}^{n/2} T''(j, m_j, m'_j)$
8: $m_i = m_i + q$; remProc = remProc $-q$
9: **end while**
10: **for** i=1 to $n/2$ **do**
11: $m'_i = \text{argmin}_{j=1}^{m_i, j|m_i} T''(i, m_i, j)$
12: **end for**
13: **OUTPUT:** $\{m_1, \ldots, m_{n/2}, m'_1, \ldots, m'_{n/2}\}$

Dividing the resources of a supercomputer between the $n/2$ instances of the k-means algorithm can be looked at as a scheduling problem, where we need to schedule $n/2$ heterogeneous tasks on a given set of resources. The tasks are heterogeneous because, for different k, k-means algorithm will do different partitioning of the data (see the previous section) which yields different degree of parallelism and different reading, computation and communication costs. Therefore, dividing the resources uniformly between the instances of the algorithm (tasks) will be sub-optimal. Furthermore, it is not possible to statically compute the precise cost of executing one instance of the algorithm on a given set of resources because, in addition to the reading (equation 1) and communication (equation 2) time that can easily be estimated, there is also a computation time that depends on the number of iteration for a particular value of k and a particular input, and this number cannot be computed statically. Therefore, we need to use some heuristics for resource allocation.

We will focus on resource allocation for level-3 partitioning, as that is the most complex of the three cases we consider. The approach we take in this paper is to use a cost function, $T''(k, m, m'_{group})$, as an estimation of the cost of executing an instance of k-means on m CPEs and m'_{group} CPE groups for each centroid. For brevity, we will annotate m'_{group} with m'. The scheduling problem can then be seen as the optimisation problem of finding the minimum of the function:

$$A(m_1, \ldots, m_{\frac{n}{2}}, m'_1, \ldots m'_{n-1}) = \sum_{i=1}^{\frac{n}{2}} T''(i, m_i, m'_i)$$

with the following constraints: $1 \leq m_i \leq p$ (for $i \in \{1, \ldots, \frac{n}{2}\}$), $\sum_{i=1}^{\frac{n}{2}} m_i \leq p, 0 \leq m'_i \leq \frac{p}{q}$ (for $i \in \{1, \ldots, \frac{n}{2}\}$), $\sum_{i=1}^{\frac{n}{2}} m'_i = \frac{p}{q}, m'_i | m_i$; (for $i \in \{1, \ldots, \frac{n}{2}\}$ where p is the total number of CPEs and q is the number of CPEs per group (64 in our case). Due to a way in which the data partitioning is done, we will require each m to cover at least one core group, i.e. to be a multiple of 64[1]. We use $T''(k, m, m') = T''_{read}(k, m, m') + T''_{comm}(k, m, m')$ as a cost function, where T''_{read} and T''_{comm} are given in the equations 1 and 2.

The algorithm that we use to solve the posed optimisation problem is given in Algorithm 4, which is based on a greedy approach. Note that, in theory, for level-3 scheduling we would need to consider allocation of individual CPEs (level-1), CGs (level-2) and CG groups (level-3) to the instances of the k-means algorithm. However, we will simplify the problem by assuming that no CG will share its resources between different instances of the algorithm. Therefore, the basic unit of allocation will be CG. The parameters of the algorithm are cost function, T'', number of available CPE groups (CGs), p, number of CPEs per CG, q, and the number of points n.

We initially allocate one CG and zero CG groups to each of the $n/2$ instances of the k-means algorithm (lines 2–4). Then, in successive iterations, we add one more CG to the instance which has the highest cost (therefore reducing its cost), until all of the CGs are allocated (lines 6–9). This, effectivelly, gives us the assignment of $m_1, m_2, \ldots, m_{n-1}$ (m_i will be the number of CGs allocated to the instance i multiplied by q (64 in our case). Once we have decided

1. In other words, we are really allocating core groups to tasks, rather than just individual CPEs

TABLE 2
Benchmarks from UCI and ImgNet

Data Set	n	k	d
Kegg Network	6.5E4	256	28
Road Network	4.3E5	10,000	4
US Census 1990	2.5E6	10,000	68
ILSVRC2012 (ImgNet)	1.3E6	160,000	196,608
ONCOLOGY and LEukemia	4.3E3	unknown	54,675

on the number of CGs for instances, we divide these CGs into CG groups, finding, for each instance, the grouping that minimised T'' (line 11). This gives us the assignment of $m'_1, m'_2, \ldots, m'_k$.

If we assume that the number of CGs is a constant, then Algorithm 4 is quadratic with respect to the number of points n. Considering that the number of points for the use cases in Section 5.2 is of the order of magnitude of 1,000,000, this is not overly expensive and can be calculated pretty quickly. The algorithm, of course, does not find an optimal allocation, as such allocation is impossible to calculate because the number of iterations that the algorithm takes for each k is not known before the execution, but it still manages to find good allocation of resources.

5 Evaluation

5.1 Experimental Design and Metrics

The datasets we applied in experiments come from well-known benchmark suites including UCI Machine Learning Repository [33] and ImgNet [22]. We briefly present the datasets in Table 2, where the first three normal size benchmarks (*Kegg Network, Road Network, US Census 1990*) are from UCI and the final high-dimensional benchmarks (*ILSVRC2012*) are from ImgNet.

The experiments have been conducted to demonstrate scalability, high performance and flexibility by increasing the number of centroids k and number of dimensions d on multiple benchmarks with vary data size n. The three-level designs are tested targeting different benchmarks. Different hardware setup will be provided for testing different scalable levels:

- *Level 1* - One SW26010 many-core processor is applied, which contains 256 64-bit RISC CPEs running at 1.45 GHz, grouped in 4 CGs in total. 64 KB LDM buffer is associated with each CPE and 32 GB DDR3 memory is shared for the 4 CGs. The theoretical memory bandwidth for register communication is 46.4 GB/s and for DMA is 32 GB/s.
- *Level 2* - Up-to 256 SW26010 many-core processors are applied, which contains 1,024 CGs in total. The bidirectional peak bandwidth of the network between multiple processors is 16 GB/s.
- *Level 3* - Up-to 4,096 SW26010 many-core processors are applied, which contains 16,384 CGs in total.

The main performance metric we are concerned with here is *one iteration completion time*. Note that the total number of iterations needed and the quality of the solution (precision) are not considered in our experiments as our work does not relate to the optimization of the underlining Lloyd algorithm or the solution of k-$means$ algorithm.

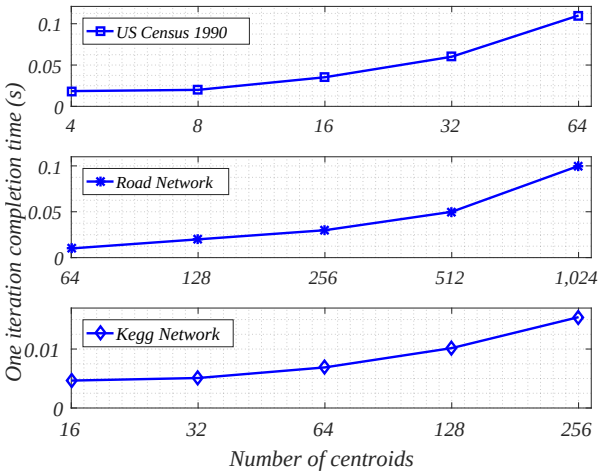


Fig. 4. Level 1 - dataflow partition

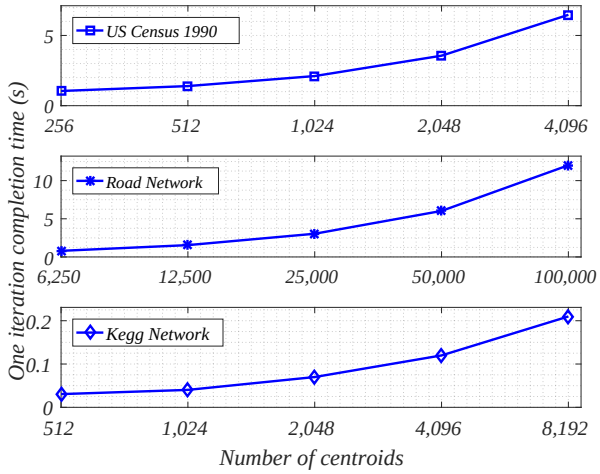


Fig. 5. Level 2 - dataflow and centroids partition

5.2 Performance and Analysis

We report the results of three different partition strategies: *Level 1* – a baseline single-level partition strategy, *Level 2* – an implementation of a state-of-the-art two-level partition strategy used in recent supercomputer implementations [3], and *Level 3* – our novel three-level partition strategy.

Since each partitioning strategy is only able to run successfully at certain ranges of k and d, it is not possible to compare them directly across the whole range benchmarks as the benchmarks have limits in terms of dataset size. For this reason, we first evaluate each strategy independently on the most suitable benchmarks for the strategy in question to show how each performs in the range for which they are most suited. The second part of our evaluation compares the partition strategies directly on benchmarks where the possible range of k and d overlap. This shows how our proposed *Level 3* strategy scales significantly better than *Level 2* over varying k, d, and number of computational nodes.

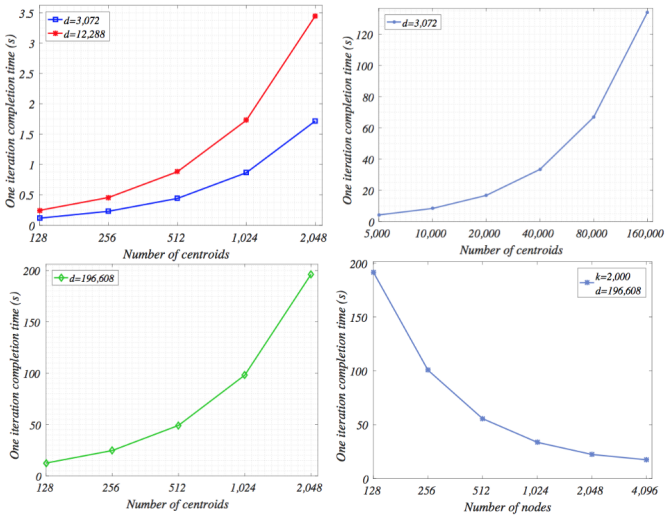


Fig. 6. Level 3 - dataflow, centroids and data-sample partition

5.2.1 Level 1 - dataflow partition

The *Level 1* (n-partition) parallel design is applied to three UCI datasets (*US Census 1990, Road Network, Kegg Network*) with their original sizes (n = 2,458,285, 434,874 and 65,554 separately) and data dimensions (d = 68, 4 and 28) for cross number of target centroids (k). The purpose of these experiments is to demonstrate the efficiency and flexibility of this approach on datasets with relatively low size, dimensions and centroid values. Figure 4 shows the *one iteration completion time* for those datasets over increasing number of clusters, k. As the number of k increases, the completion time on this approach grows linearly.

5.2.2 Level 2 - dataflow and centroids partition

The level 2 (nk-partition) parallel design is applied to same three UCI datasets as above, but for a large range of target centroids (k). The purpose of these experiments is to demonstrate the efficiency and flexibility of the proposed approaches on datasets with large-scale target centroids (less than 100,000). Figure 5 shows the *one iteration completion time* of the three datasets of increasing number of clusters, k. As the number of k increasing, the completion time from this approach grows linearly. We conclude that this approach works well when one dimension is varied up to the limits previously published.

5.2.3 Level 3 - dataflow, centroids and dimensions partition

The *Level 3* (nkd-partition) parallel design is applied to a subset of ImgNet datasets (*ILSVRC2012*) with its original size (n = 1,265,723). The results are presented with varying number of target centroids (k) and data dimension size (d) with an extremely large domain. We also test the scalability varying the number of computational nodes. The purpose of these experiments is to demonstrate the high performance and scalability of the proposed approaches on datasets with large size, extremely high dimensions and target centroids. Left hand side of Figure 6 shows the completion time of the dataset of increasing number of clusters, k = 128, 256, 512, 1024 and 2,048 with increasing number of dimensions, d = 3,072 (32*32*3), 12,288 (64*64*3) and 196,608 (256*256*3).

To further investigate the scalability of our approach, we test two more cases by either further scaling centroids by certain number of data dimensions (d = 3,072) and number of nodes ($nodes$ = 128) or further scaling nodes applied by certain number of data dimensions (d = 196,608) and number of centroids (k = 2,000). The results of those two tests are shown in the right hand side of Figure 6.

As both k and d increase, the completion time from our approach continues to scale well, demonstrating our claimed high performance and scalability.

5.2.4 Comparison of partition levels

In this section we experimentally compare the *Level 2* approach with *Level 3*.

Figure 7(1) shows how *one iteration completion time* grows as the number of dimensions increases. The Level 2 approach outperforms Level 3 when the number of dimensions is relatively small. However, the Level 3 approach scales significantly better with growing dimensionality, outperforming Level 2 for all d greater than 2560. The Level 2 approach cannot run with d greater than 4096 in this scenario due to memory constraints. However, it is clear that, even if this problem were solved, the poor scaling would still limit this approach. The completion time for Level 2 falls twice unexpectedly between 1536 and 2048, and between 2560 and 3072. This is due to the crossing of communication boundaries in the architecture of the supercomputer – the trend remains clear however.

Figure 7(2) shows how the *one iteration completion time* grows as the number of centroids, k increases. Since the number of d is fixed at 4096, the Level 3 approach actually always outperforms Level 2, with the gap increasing as k increases. This scaling trend is replicated at lower levels of d too, though Level 2 initially outperforming Level 3 at lower values of k.

Figure 7(3) shows how both Level 2 and Level 3 scale across an increasing number of computation nodes. Level 3 clearly outperforms Level 2 in all scenarios. The values of k and d are fixed, as described in the graph caption, at levels which Level 2 can operate. The performance gap narrows as more nodes are added, but remains significant. Clearly the exact performance numbers will vary with other values k and d, as can be inferred from other results, but the main conclusion we draw here is that Level 3 generally scales well.

5.2.5 Comparison with other architectures

As discussed, state-of-the-art supercomputing-oriented approaches are tested either on their specific datasets [7], [26] or publish only their relative speedups [3] instead of execution times. It is not possible to compare our actual execution time with these supercomputing-oriented approaches directly. Additionally, wallclock execution times are problematic to compare across vastly differing architectures with different budgets.

To give some insight into the performance we obtain, we compare execution time with other architectures directly where this is possible. We present five comparable results from published literature in Table 3. Based on the differing workload sizes presented in these papers, we adjust the hardware configuration for Sunway TaihuLight, changing

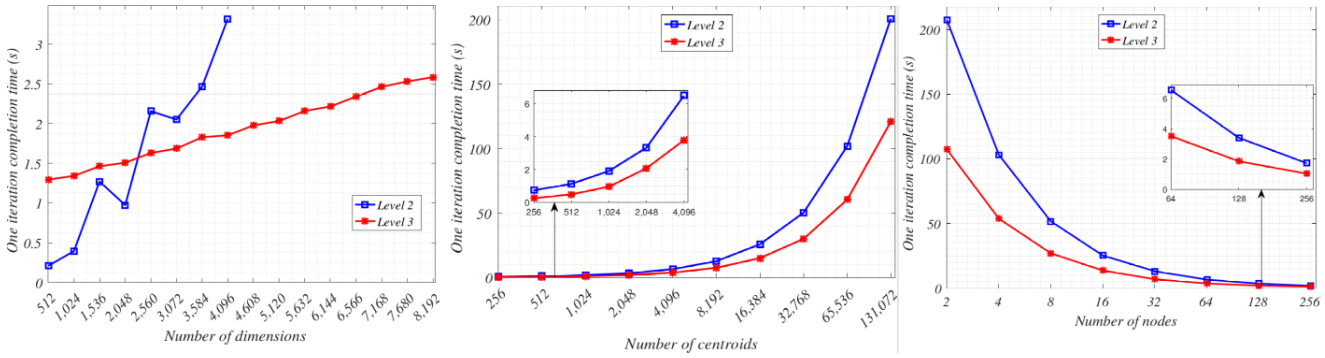


Fig. 7. Comparison tests: (1) varying d with 2,000 centroids and 1,265,723 data samples tested on 128 nodes; (2) varying k with 4,096 dimensions and 1,265,723 data samples tested on 128 nodes; (3) varying number of nodes used with a fixed 4,096 dimension, 2,000 centroids and 1,265,723 data samples.

TABLE 3
Execution time comparison with other architectures

Approaches	Hardware Resources	n	k	d	Execution time per iteration (sec.)	Execution time per iteration by Sunway Taihu-Light (sec.)	Max. Speedup
Rossbach, et al [34]	10x NVIDIA Tesla K20M + 20x Intel Xeon E5-2620	1.0E9	120	40	49.4	0.468635 (128 nodes)	105x
Bhimani, et al [4]	NVIDIA Tesla K20M	1.4E6	240	5	1.77	0.025336 (4 nodes)	70x
Jin, et al [25]	NVIDIA Tesla K20c	1.4E5	500	90	5.407	0.110191 (1 node)	49x
Li, et al [28]	Xilinx ZC706	2.1E6	4	4	0.0085	0.002839 (1 node)	3x
Ding, et al [15]	Intel i7-3770K	2.5E6	10,000	68	75.976	2.424517 (16 nodes)	31x

the number of nodes utilized. This is determined by the size of the task in terms of k and d where no further performance gains are possible by adding more nodes. The number of nodes varies from just one node for a single processing unit [25], [28] to 128 nodes in [34]. We report results against a heterogeneous node based approach running a custom implementation of parallel k-means on ten heterogeneous nodes, each node consisting of an NVIDIA Tesla K20M GPU with two Intel Xeon E5-2620 CPUs [34]. Further, we compare against two GPU based implementations running on an NVIDIA Tesla K20M GPU and an NVIDIA Tesla K20C GPU respectively [4], [25], an FPGA based approach running a custom parallel k-means implementation on Xilinx ZC706 FPGA [28], and a multi-core processor based approach running a custom implementation of parallel k-means on 8-core Intel i7-3770k processor [15].

The proposed approach running on the Sunway Taihu-Light supercomputer achieves more than 100x speedup over the high-performance heterogeneous nodes based approach, between 50x-70x speedup than those single GPU based approaches, and 31x speedup over multi-core CPU based approach on their largest solvable workload sizes.

5.3 Auto-clustering on Real Application

Genomic information from gene expression data has been widely used and already benefited on improving clinical decision and molecular profiling based patient stratification. Clustering methods, as well as their corresponding HPC-based solutions [39], are adopted to classify the high-dimensional gene expression sequences into some known patterns, which indicates that the number of targeted clustering centroids are determined in advance. As we all know, there are still large numbers of gene expression sequences, among which the patterns are not yet discovered. Therefore, the proposed auto-clustering method can potentially help find new patterns from high-dimensional gene expression datasets.

In our work, we test the auto-clustering process on the ONCOLOGY&LEukemia gene expression datasets [16]. There are 4254 subjects and each subject has 54675 probesets. In this problem definition, we cluster the whole dataset using our level-3 partitioning method, where n is 4254, and d is 54675. In this task, we generate the candidate k by enumerating from 2 to 2000 (up-to around $n/2$). The performance for one iteration execution time is shown in figure 8(1) and the total execution time is shown in figure 8(2). The results demonstrate good performance of our approach with a linear scale on one iteration time and also shows that our supercomputer-based technique can compute such a large-scale dataset for all needed iterations within 200 seconds at most.

We further apply the evaluation function to determine the optimal value of k. The results are shown in figure 8(3). We can see that $r'(k)$ reaches the first increasing when $k = 14$. After that, $r'(k)$ fluctuates around a certain value, which indicates that continually increasing the k values cannot further represent more patterns in the input data.

6 Conclusion

In this paper, we present an automatic k-means clustering solution based on the Sunway TaihuLight supercomputer.

We first propose a fully data partitioned (nkd-partition) approach for parallel k-$means$ implementation to achieve scalability and high performance at large numbers of centroids and high data dimensionality simultaneously. Run-

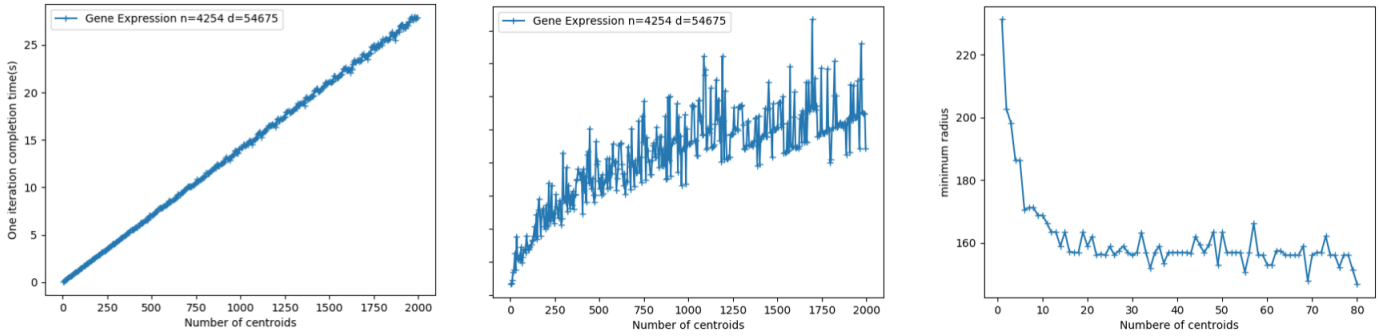


Fig. 8. (1) One iteration execution time and (2) total execution time for gene expression dataset ONCOLOGY and LEukemia; (3) The evaluation function $r'(k)$ to determine the optimal k value.

ning on the Sunway TaihuLight supercomputer, it breaks previous limitations for high performance parallel k-$means$.

Furthermore, we propose an automatic hyper-parameter determination process, by automatically generating and executing the clustering tasks with a number of candidate hyper-parameters, and then determining the optimal hyper-parameter according to an evaluation method.

The proposed auto-clustering solution is a significant attempt to support *AutoML* on a supercomputer system, and provide a feasibles way to support other potential machine learning algorithms.

Acknowledgment

This work is partially funded by the UK EPSRC grant *Discovery: Pattern Discovery and Program Shaping for Manycore Systems* (grant code EP/P020631/1).

References

[1] Mihai Bādoiu, Sariel Har-Peled, and Piotr Indyk. Approximate clustering via core-sets. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 250–257. ACM, 2002.

[2] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of computational biology*, 6(3-4):281–297, 1999.

[3] Michael A Bender, Jonathan Berry, Simon D Hammond, Branden Moore, Benjamin Moseley, and Cynthia A Phillips. k-means clustering on two-level memory systems. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 197–205. ACM, 2015.

[4] Janki Bhimani, Miriam Leeser, and Ningfang Mi. Accelerating k-means clustering with parallel implementations and gpu computing. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6. IEEE, 2015.

[5] Christian Böhm, Martin Perdacher, and Claudia Plant. Multi-core k-means. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 273–281. SIAM, 2017.

[6] Thomas Bottesch, Thomas Bühler, and Markus Kächele. Speeding up k-means by approximating euclidean distances via block vectors. In *International Conference on Machine Learning*, pages 2578–2586, 2016.

[7] Y Dora Cai, Rabindra Robby Ratan, Cuihua Shen, and Jay Alameda. Grouping game players using parallelized k-means on supercomputers. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 10. ACM, 2015.

[8] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33(6):1417–1440, 2004.

[9] Siya Chen, Tieli Sun, Fengqin Yang, Hongguang Sun, and Yu Guan. An improved optimum-path forest clustering algorithm for remote sensing image segmentation. *Computers & Geosciences*, 112:38–46, 2018.

[10] Guy Barrett Coleman and Harry C Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

[11] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li, and Changqing Ji. Optimized big data k-means clustering using mapreduce. *The Journal of Supercomputing*, 70(3):1249–1259, 2014.

[12] Ryan R Curtin. A dual-tree algorithm for fast k-means clustering with large k. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 300–308. SIAM, 2017.

[13] Sanjoy Dasgupta. *The hardness of k-means clustering*. Department of Computer Science and Engineering, University of California, San Diego, 2008.

[14] Inderjit S Dhillon and Dharmendra S Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-scale parallel data mining*, pages 245–260. Springer, 2002.

[15] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *International Conference on Machine Learning*, pages 579–587, 2015.

[16] Expression Project for Oncology (expO). http://www.intgen.org/.

[17] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[18] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE transactions on knowledge and data engineering*, 15(3):515–528, 2003.

[19] Ali Hadian and Saeed Shahrivari. High performance parallel k-means clustering for disk-resident datasets on multi-core cpus. *The Journal of Supercomputing*, 69(2):845–863, 2014.

[20] Greg Hamerly. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 130–140. SIAM, 2010.

[21] Juan Mario Haut, Mercedes Paoletti, Javier Plaza, and Antonio Plaza. Cloud implementation of the k-means algorithm for hyperspectral image analysis. *The Journal of Supercomputing*, 73(1):514–529, 2017.

[22] ImgNet ILSVRC2012. http://www.image-net.org/challenges/lsvrc/2012/.

[23] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[24] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: a survey. *IEEE Transactions on knowledge and data engineering*, 16(11):1370–1386, 2004.

[25] Yu Jin and Joseph F Jaja. A high performance implementation of spectral clustering on cpu-gpu platforms. *arXiv preprint arXiv:1802.04450*, 2018.

[26] Jitendra Kumar, Richard T Mills, Forrest M Hoffman, and William W Hargrove. Parallel k-means clustering for quantitative ecoregion delineation using large data sets. *Procedia Computer Science*, 4:1602–1611, 2011.

[27] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 115–122. IEEE, 2010.

[28] Zhehao Li, Jifang Jin, and Lingli Wang. High-performance k-means implementation based on a simplified map-reduce architecture. *arXiv preprint arXiv:1610.05601*, 2016.

[29] Top500 list. https://www.top500.org/lists/2018/06/.

[30] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[31] James Newling and François Fleuret. Fast k-means with accurate bounds. In *International Conference on Machine Learning*, pages 936–944, 2016.

[32] James Newling and François Fleuret. Nested mini-batch k-means. In *Advances in Neural Information Processing Systems*, pages 1352–1360, 2016.

[33] UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets.html.

[34] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[35] Xiao-Bo Shen, Weiwei Liu, Ivor W Tsang, Fumin Shen, and Quan-Sen Sun. Compressed k-means for large-scale clustering. In *AAAI*, pages 2527–2533, 2017.

[36] Rui Shi, Huamin Feng, Tat-Seng Chua, and Chin-Hui Lee. An adaptive image content representation and segmentation approach to automatic image annotation. In *International conference on image and video retrieval*, pages 545–554. Springer, 2004.

[37] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

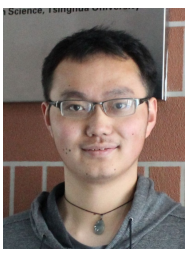
[38] Leonardo Torok, Panos Liatsis, Jos Viterbo, Aura Conci, et al. k-ms. *Pattern Recognition*, 66(C):392–403, 2017.

[39] Shicai Wang, Ioannis Pandis, David Johnson, Ibrahim Emam, Florian Guitton, Axel Oehmichen, and Yike Guo. Optimising parallel r correlation matrix calculations on gene expression data using mapreduce. *BMC bioinformatics*, 15(1):351, 2014.

[40] Mario Zechner and Michael Granitzer. Accelerating k-means on the graphics processor via cuda. In *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*, pages 7–15. IEEE, 2009.

[41] Dengsheng Zhang, Md Monirul Islam, and Guojun Lu. A review on automatic image annotation techniques. *Pattern Recognition*, 45(1):346–362, 2012.

Teng Yu is a PhD candidate in School of Computer Science, University of St Andrews. He received a MRes in Advanced Computing from Imperial College London in 2016, a BSc in Computer Science from University College Cork joint with Beijing Technology and Business University in 2015. His research interests include operating systems, heterogeneous architectures and HPC.

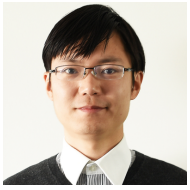


Wenlai Zhao is a PostDoctoral researcher in Tsinghua University and is leading the distributed machine learning research group in the National Supercomputing Center in Wuxi (NSCCWX). He got his Ph.D. degree from Department of Computer Science and Technology in Tsinghua University. His research interest is heterogeneous parallel computing and distributed machine learning.

Pan Liu is a final year undergraduate student in Department of Computer Science and Technology, Tsinghua University. His research interests include HPC and cryptography.

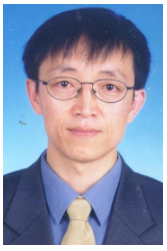
Vladimir Janjic is a PostDoctoral researcher in the School of Computer Science, University of St Andrews. His research interests are high-level programming models (based on the concept of parallel patterns) for heterogeneous parallel platforms, as well as efficient implementation of these models on hardware.

Xiaohan Yan is a PhD candidate in Department of Mathematics, University of California, Berkeley. His research interests lie generally in the intersection of symplectic geometry, algebraic geometry and mathematical physics, including quantum K-theory, moduli space, and mirror symmetry.



Shicai Wang is a Bioinformatician in Wellcome Trust Sanger Institute. He has a PhD in computing from Imperial College London. His research interests include high performance computing, databases and bioinformatics.

Haohuan Fu is a professor in the Department of Earth System Science in Tsinghua University, where he leads the research group of High Performance Geo-Computing (HPGC). He is also the deputy director of the National Supercomputing Center in Wuxi, leading the R&D division. He got a PhD from Imperial College London. He has been working towards the goal of providing both the most efficient simulation platforms and the most intelligent data management and analysis platforms for geoscience applications.



Guangwen Yang is a professor in the Department of Computer Science in Tsinghua University, and is the director of the National Supercomputing Center in Wuxi. He got the PhD from Harbin Institute of Technology. His research interests include distributed system, parallel computing and machine learning system.

John Thomson is a Lecturer (Assistant Professor) in the School of Computer Science, University of St Andrews. John's Research interests center around empirical approaches to systems design, including optimising compilers, video compression, HPC, embedded systems, applied machine learning, parallelisation techniques and runtime systems.